Evaluation of Hotlink Assignment Heuristics for Improving Web Access

Jurek Czyzowicz [§‖] Evangelos Kranakis [†‡‖] Danny Krizanc [¶‡‖] Andrzej Pelc [§‖]
Miguel Vargas Martin [†‡*]

Abstract

We study the optimal hotlink assignment problem. Consider a web site as a directed graph, where each page is represented by a node and each link is represented by an edge. The resulting graph is connected and may have cycles. It was previously proven that the hotlink assignment problem is NP-hard in its full generality. We describe a way of constructing a tree that maintains the original semantic relationship between the pages of a web site. We then proceed to describe three algorithms that improve access in web site trees. We simulate our algorithms in three different, but closely related, kinds of trees: power-law trees, realistic trees, and an actual tree. These algorithms are capable of obtaining considerable savings on the access cost of the web site. **Key words and phrases:** web sites structure, web sites access cost, hotlinks, web sites performance, simulations, random trees, random web sites.

[†]School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. E-mail: {kranakis,mvargas}@scs.carleton.ca.

[¶]Department of Mathematics, Wesleyan University, Middletown CT 06459, USA. Email: dkrizanc@caucus.cs.wesleyan.edu.

[§]Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada. E-mail: pelc@uqah.uquebec.ca.

[‖]Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

[‡]Research supported in part by MITACS (Mathematics of Information Technology and Complex Systems) NCE (National Center of Excellence) grant.

[*]Research supported in part by CONACYT (Science and Technology Coucil of Mexico).

1 Introduction

The irregular growth of the Internet has resulted in a huge and complex network that can be partially modeled as a directed graph, whose nodes are documents and edges are links (URLs) pointing from one document to another. The graph's topology determines the web's connectivity from which the efficiency in locating information is derived.

The question of what does the Internet look like, is, a complicated problem and has given rise to numerous experimental studies concerning the true structure of the web. In a survey of web characterization, Pitkow [17] analysed the web from four points of view: client (user) behaviour, proxies and gateways, servers, and the whole web. An important consequence of proxies and gateways analysis was that URL access follows a Zipf's distribution (see [12] for a description of Zipf's probability distribution). Zipf's law was also shown to govern server access. Abrams et al. [2] obtained important statistics about how people use bookmarks. Tauscher et al. [20] obtained interesting results of how users revisit a web page. Reka et al. [19] found that the diameter[*] of the web is 19. Faloutsos et al. [7] found that the web follows a power-law on the nodes degree (the number of nodes with degree k is given by k^{-g}, where g is a constant). Related studies seeking to extrapolate from the structure of the telephone network have also been conducted, see for example [1].

One of the most widely proposed solutions for improving web access is caching. Caching consists of storing the most requested web pages in the proxy

[*]Diameter is defined as the average number of hops between two random pages.

for future access (see [14] for an overview of proxies). Many caching techniques have been proposed. Pitkow et al. [18] present an adaptive caching algorithm based on the analysis of a psychological model of human memory that responds to the hit rates and access patterns of users requesting documents. Glassman [8] describes the design and performance of a proxy based on a study on human behaviour. Li et al. [13] studied the optimal placement of web proxies in line and tree topologies.

More recently, attention has been shifted toward the optimal design of web sites. Some authors ([5, 6, 15]) suggest analyzing user access patterns to help design better web pages, sites, and browsers. Perkowitz et al. [15] proposed the design of "adaptive web sites" by promoting and demoting pages, highlighting links, adding hotlinks, and clustering related pages. Two years later, Perkowitz et al. [16] presented the PageGater algorithm, which analyzes user access logs in order to identify candidate link sets to be included in index pages. Their algorithm performs the following steps: a) process the access log into visits, b) find clusters of linked pages and create an adjacency matrix, c) find maximal cliques, d) rank the cliques found, and e) for each cluster, create a web page consisting of links to the documents in the cluster.

We propose an improvement on web site access by making the most popular pages more accessible to users. Suppose that there is a page with high access hits; we want this page to be closer to the home page so users can reach it in less hops. We do this by adding hotlinks to the existing structure of the web site.

2 Notation and terminology

Consider a part of the web called a *web site*, consisting of a collection $V = \{v_1, \ldots, v_n\}$ of web pages connected by links. These links have been placed a priori by design in the initial construction of the web pages. Assume that there exists a directed path of links from the *home page* r to any other page of the collection. We can view the web site as a directed graph $G(V, E)$, where each page is represented by a node, and each link is represented by an edge.

Consider a tree $T(V, E')$, where $E' \subseteq E$, with a distinguished node called the root, r, and a probability distribution over its leaves. Let us say leaf i has probability (weight) p_i. We assign weights to the intermediate leaves in a bottom-up fashion, in such a way that the weight of a node, p_i, is equal to the sum of the probabilities of the leaves descendant to it.

We define the *distance* from a node $v \in V$ to the root r, denoted by $d(v)$, as the number of edges between them. The *access cost* of leaf v, denoted as $c(v)$, is defined by

$$c(v) = p_v d(v)$$

Thus, the *expected number of steps* to reach a leaf from the root is defined by

$$E[T] = \sum_{v \text{ is a leaf}} c(v) \tag{1}$$

Our problem is to minimize equation 1 by adding hotlinks to the tree. A *hotlink, h,* is an additional directed edge, $(s,t)^\dagger$, added to the original tree, such that t is a descendant of s. We say that s is the *hyperparent* of t, and t is the *hyperson* of s. Consider a set $H = \{h_1, \ldots, h_k\}$ of hotlinks assigned to the tree T; the resulting graph is denoted by T^H, and the gain of H is defined by

$$G(H) = E[T] - E[T^H]$$

The gain of a single hotlink, $h = (s,t)$, is defined by

$$g(h) = p_t((d(t) - d(s) - 1)) \tag{2}$$

The *hotlink assignment to arbitrary graphs* problem was proven to be NP-hard in [3].

3 Results of the paper

We describe a procedure for creating a naturally embedded tree within a web site. This tree maintains the structure of the site conceived by its designer.

†Literals s and t stand for *source* and *target* nodes.

heuristic	power-law	realistic	actual
greedyBFS	25.63	16.04	27.25
recursive	27.06	17.49	28.29
2-greedyBFS	39.40	23.25	50.27

Table 1: *Average highest percentage of savings obtained by applying our algorithms to three different kinds of trees.*

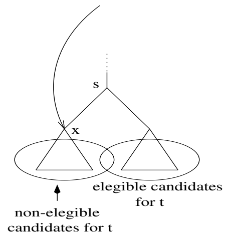

Figure 1: *In* greedyBFS *algorithm, we do not allow two hotlinks to cross each other. Consider a hotlink* (s,t) *to be assigned in an iteration of the algorithm.* t *must be a descendant of* s *that maximizes the gain, but that is not descendant of a node* x*, which is at a higher level than* s *and already has an incoming hotlink.*

We introduce three heuristic algorithms as an approach to the optimal hotlink assignment problem. We then proceed to simulate our algorithms on three types of trees: power-law trees with Zipf's distribution, real trees with Zipf's distribution, and an actual web site tree with known probability distribution. The results of the simulation reveal that our algorithms can considerably reduce the access cost of a web site. Table 1 shows the percentage of savings achieved by our algorithms.

4 Hotlink optimization heuristics

We can look at a web site as a directed graph, where each page is represented by a node, and each link is represented by an edge. The resulting graph is connected, and may have cycles. The study of hotlink assignments within this graph is a complicated task. In fact, Bose et al.[3] have proven that the hotlink assignment problem is NP-hard. Fortunately, this graph contains a naturally embedded tree that can be constructed starting from the home page of the site. The home page has links to internal pages that also have links to other internal pages and so on. This perception of the web site gives rise to a Breadth First Search (BFS) tree with the home page as the root. Such a tree maintains the semantic relationship between web pages in such a way that related pages have a common ancestor. Moreover, the tree respects the most natural and basic structure originally conceived by the web site designer.

In this section we introduce three hotlink heuristic optimization algorithms. Our first two algorithms are BFS oriented, and the third one is recursive. We assign at most one hotlink per node.

The simplest algorithm is called *simpleBFS*. This algorithm assigns hotlinks iteratively in a BFS order, starting from the root node. Let $A = \{v_1, v_2, ..., v_n\}$ be the set of nodes of the tree in increasing BFS order in such a way that v_1 is the root. Now in the *i-th* iteration of the algorithm, we assign a hotlink (s,t), where $s = v_i$, and t is a descendant of s that maximizes the gain. The algorithm stops when there are no more possible hotlinks to assign.

The next algorithm that we present is a slight modification of simpleBFS. Here, we also assign hotlinks iteratively in a BFS order starting from the root. The only difference is that in each iteration of the algorithm, we do not allow two hotlinks to cross each other, i.e., t is a descendant of s that maximizes the gain, but that is not a descendant of a node x, which is at a higher level than s and already has an incoming hotlink (see figure 1). The algorithm stops when there are no more possible hotlinks to assign. We call this algorithm *greedyBFS* because certainly none of the hotlinks is waisted, as we will see.

Our third algorithm is a recursive version of *greedyBFS*. The basic idea is to assign a hotlink (s,t), where s is the root and t is the node that offers the biggest gain, then split the original tree in subtrees consisting of the subtrees rooted at the children of s and the subtree rooted at t, and proceed recursively for each of these subtrees (see figure 4). We simply call this algorithm *recursive*.

4.1 simpleBFS algorithm

The basic idea of this algorithm is to iteratively assign hotlinks in a BFS order. This algorithm assigns hotlinks iteratively in a BFS order, starting from the root node. Consider a hotlink (s, t), in each iteration of the algorithm, s corresponds to the next node in BFS order and t corresponds to a descendant of s that offers the biggest gain. The algorithm stops when there are no more possible hotlinks to assign. simpleBFS algorithm uses the function *nextBFS()*, which returns one by one the nodes of the tree in a breadth first search order, starting from the root.

simpleBFS(T)

1. $H = \phi$
2. *while((s = nextBFS()) $\neq \phi$)*
   (a) *$t = v : v$ maximizes equation 2; and v is descendant of s; and v does not have a hyperparent*
   (b) *if $t \neq \phi$ then $H = H \bigcup \{(s,t)\}$*

Loop of step 2 is executed $O(\mid V \mid)$ times, and step 2a has a running time of $O(\mid V \mid)$. Therefore, the total running time of the algorithm is $O(\mid V \mid^2)$.

A variant of *simpleBFS* algorithm consists of not allowing the target node to be a descendant of a node that already has a hyperparent. We call this algorithm *greedyBFS*. Note that the only modification takes place in step 2a. The algorithm's complexity remains the same.

4.2 greedyBFS algorithm

greedyBFS assigns hotlinks iteratively in a BFS order starting from the root. Consider a hotlink (s, t), in each iteration of the algorithm, s corresponds to the next node in BFS order, and t corresponds to the descendant of s that maximizes the gain, but that is not descendant of a node x, which is at a higher level than s and already has an incoming hotlink. The algorithm stops when there are no more possible hotlinks to assign.

greedyBFS(T)

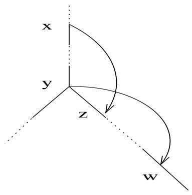

Figure 2: *Illustration of* "obvious navigation" *assumption. Let us say a user is at page x and wishes to reach a page descendant of w. We assume that, as the user does not have a map of the site, the path $(x, z) + z \to w$ will be followed, even though path $x \to y + (y, w)$ had been shorter. Therefore, obvious navigation is efficient only when $x \to y \geq z \to w$.*

1. $H = \phi$
2. *while((s = nextBFS()) $\neq \phi$)*
   (a) *$t = v : v$ maximizes equation 2; and v is a descendant of s; and v does not have a hyperparent; and v is not a descendant of a node x, which is at a higher level than s and already has a hyperparent.*[‡]
   (b) *if $t \neq \phi$ then $H = H \bigcup \{(s,t)\}$*

By comparing these two algorithms we can prove that *greedyBFS* performs at least as well as simpleBFS.

Theorem 1. *Consider an arbitrary tree T with arbitrary probability distribution over its leaves. Let H be the set of hotlinks resulting from running* greedyBFS *on T, and let H' be the set of hotlinks resulting from running simpleBFS on T, then $G(H) \geq G(H')$. In other words,* greedyBFS *performs at least as well as simpleBFS.*

Proof. In order to prove the theorem we need one more definition. We assume users perform *obvious navigation*. We say obvious navigation consists of taking always the link or hotlink that will take us closer to the desired page. Note that in some cases this kind of navigation can be inefficient (see figure 2).

[‡] We say that the levels of the tree are counted in increasing order starting from the root, in such a way that the root is at the lowest level.

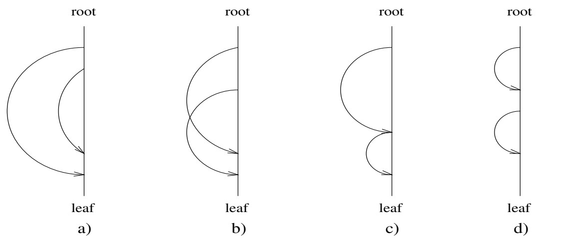

Figure 3: *Four possible scenarios of two hotlinks on a path, after executing simpleBFS algorithm. Each figure shows an arbitrary path from the root to a leaf. a) one hotlink is "inside" the other, b) the hotlinks are overlapped, c) one hotlink starts exactly at the end of the other, d) the hotlinks do not touch.*

Now, we will show that *greedyBFS* performs better than simpleBFS. If we took two hotlinks from an arbitrary path from the root to a leaf after running simpleBFS, we could find any of the following four cases: a) one hotlink is "inside" the other, or b) two hotlinks are overlapped, or c) one hotlink starts exactly at the end of the other, or d) the hotlinks do not touch. These four cases are shown respectively in figure 3.

It is easy to see that obvious navigation can be inefficient only in case b). Even more, in this case the deepest hotlink will be waisted, disabling the source node to accommodate an efficient hotlink from it. Case b) is unlikely to happen in *greedyBFS* algorithm. Therefore, the total gain obtained by *greedyBFS* is always better or equal than the one obtained by *simpleBFS*.

□

4.3 recursive algorithm

The basic principle of this algorithm is to assign a hotlink (s,t), where s is the root and t is the node that maximizes equation 2, then split the original tree in subtrees consisting of the subtrees rooted at the children of s and the subtree rooted at t, and proceed recursively for each of these subtrees. Note that one of the children of s, s_k, will be ancestor of t, and the subtree rooted at t will not form part of the subtree rooted at s_k. Figure 4 illustrates the algorithm. In our algorithm, we assume that the set of hotlinks, H, is initially empty.

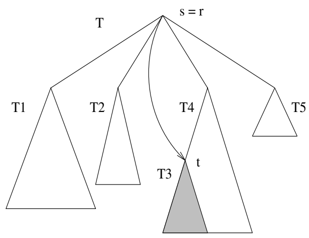

Figure 4: *First call to* recursive(T). *The algorithm assigns a hotlink from the root s to the node t that maximizes equation 2 and then proceeds recursively for T_i, $i = 1, ..., 5$. Note that T_3 is not part of T_4 and they are treated as separate trees.*

recursive(T)

1. $s = r$
2. $t = v$: *v maximizes equation 2; and is not a descendant of a node x, which is at a higher level than s and already has a hyperparent*
3. *if* $t \neq \phi$ *then* $H = H \bigcup \{(s,t)\}$
4. *for each children of s, s_i (including t)*
   (a) T = *subtree rooted at* s_i
   (b) recursive(T)

The algorithm makes $O(|\ V\ |)$ recursive calls, each of them having $O(|\ V\ |)$ running time complexity. Therefore, the overall complexity of the algorithm is $O(|\ V\ |^2)$.

4.4 k-algorithms

Here we are allowed to assign at most k outgoing hotlinks per node. We present the corresponding modified versions of *greedyBFS* and *recursive*. Again, the function *nextBFS()* returns one by one the nodes of the tree in a breadth first search order, starting from the root.

k-**greedyBFS(** T, k)

1. $H = \phi$
2. *while((s = nextBFS()) $\neq \phi$)*
   (a) *for* $j = 1$ *to* k

i. *$t = v : v$ maximizes equation 2; and v is a descendant of s; and v does not have a hyperparent; and v is not descendant of a node x, which is at a higher level than s and already has a hyperparent*

ii. *if $t \neq \phi$ then $H = H \bigcup \{(s,t)\}$*

The loop of step 2 is repeated $O(\mid V \mid)$ times, while the step 2(a)i takes $O(\mid V \mid)$ running time. The total running time complexity of the algorithm is $O(\mid V \mid^2)$.

We also suggest a recursive version of the *recursive* algorithm:

***k*-recursive(** T, k)

1. *$s = r$*
2. *for $i = 1$ to k*
   (a) *$t = v : v$ maximizes equation 2; and v is not a descendant of a node x, which is at a higher level than s and already has a hyperparent; and v has no hyperparent*
   (b) *if $t \neq \phi$ then $H = H \bigcup \{(s,t)\}$*
3. *for each children of s, s_i (including hypersons)*
   (a) *T = subtree rooted at s_i*
   (b) k-recursive(T, k)

5 Simulations

In this section we describe the process of our simulations and present the results obtained. All our simulations were implemented with *Java++* 6.0. We evaluated our algorithms on three different kinds of structures: power-law trees with Zipf's distribution, real trees with Zipf's distribution, and an actual tree with known probability distribution.

Zipf's distribution governs web site access [17]. Zipf's distribution is defined as follows [12]:

$$p_i = \frac{1}{iH_n},$$

where H_n is the harmonic number:

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

In our first two experiments, we distributed the p_i values over the n leaves of the tree in a random fashion.

5.1 Power-law trees with Zipf's distribution

Faloutsos et al. [7] discovered that the Internet topology follows power-law relationships. In particular, we are interested in the power-law that governs the degree of a node: the probability that a node has degree i is proportional to i^{-k}, for $k > 1$. Broder et al. [4] concluded that $k = 2.72$ is an accurate value for the outdegree of a node. Now we will describe the process of our simulation:

1. For $n = 1000$ to $n = 17000$ increment n by 1000:
   (a) Generate 100 random power-law trees $T(V,E)$ of order $\mid V \mid= n$ based on the outdegree power-law: the probability that a node has i children is proportional to $i^{-2.72}$. The maximum number of children per node is bounded to 10. For each of these 100 trees do the following:
   (b) Assign a Zipf's probability distribution over the leaves. This probability represents the access probability of the leaves. Then, assign access probabilities to the intermediate nodes in a bottom-up fashion, in such a way that the probability of a node is equal to the sum of the probabilities of the leaves descendant to it.
   (c) Compute the expected number of steps to reach a leaf from the root $E[T]$.
   (d) Run *greedyBFS*, *k-greedyBFS*, and *recursive* algorithms on the tree.
      i. $H_g = greedyBFS(T)$
      ii. $H_r = recursive(T)$
      iii. $H_k =$*k-greedyBFS(T, 2)*

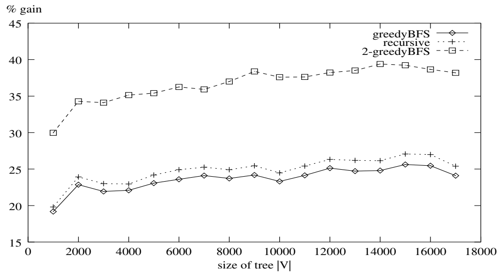

Figure 5: *Gain obtained by applying* greedyBFS*,* recursive*, and* 2-greedyBFS *to random power-law trees of size* $|V| = n$ *(for* $n = 1000 \ldots 17000$*) with Zipf's probability distribution over the leaves.*

heuristic	min gain %	max gain %
greedyBFS	19.18	25.63
recursive	19.80	27.06
2-greedyBFS	29.97	39.40

Table 2: *Minimum and maximum gains obtained by applying* greedyBFS*,* recursive*, and* 2-greedyBFS *to random power-law trees of size* $|V| = n$ *(for* $n = 1000 \ldots 17000$*) with Zipf's probability distribution over the leaves.*

(e) Compute the expected number of steps to reach a leaf from the root $E[T^{H_g}]$, $E[T^{H_r}]$, and $E[T^{H_k}]$.

(f) Compute the gain $G(H)$ obtained by each of these algorithms.

i. $G(H_g) = E[T] - E[T^{H_g}]$
ii. $G(H_r) = E[T] - E[T^{H_r}]$
iii. $G(H_k) = E[T] - E[T^{H_k}]$

Figure 5 illustrates the results of our experiments. Note that the gain of each algorithm oscillates between narrow intervals, which indicates that the size of the tree does not greatly affect the performance. Our results are shown in table 2. The *k-greedyBFS* algorithm performs better than the others because the maximum number of outgoing hotlinks allowed per node is bigger.

5.2 Real trees with Zipf's distribution

There are many techniques for generating random graphs, see for example [21, 10, 11]. Even more, Broder et al. [4] found accurate parameters for the power-law relationships between the degree of a node and its frequency. However, these power-laws hold for graphs, not for trees. So far we do not know of any precise technique for generating web site trees. Therefore, we have collected the link structure[§] of the web sites of twelve Ontario Universities[¶] on which to test our algorithms. Now we describe our experiments:

1. Explore the web site and obtain its html, htm, shtml, and shtm files.
2. Look at each of the files (only html, htm, shtml, and shtm files) referenced in the home page. For each of these files, repeat in a BFS order until the entire file structure has been visited. This BFS traversal gives rise to a BFS tree of the web site.
3. Perform 100 random assignments of Zipf's probability distributions to the leaves. This probability represents the access probability of the leaves. Then, assign access probabilities to the intermediate nodes in a bottom-up fashion, in such a way that the probability of a node is equal to the sum of the probabilities of the leaves descendant to it. For each of these 100 weighted trees do the following:
4. Compute the expected number of steps to reach a leaf from the root $E[T]$.
5. For $i = 1$ to 16384 increment i by $i \times 2$

   (a) Run *greedyBFS*, *k-greedyBFS*, and *recursive* algorithms on the tree with the restriction that $|H| \leq i$.

   i. $H_g = greedyBFS(T)$
   ii. $H_r = recursive(T)$
   iii. $H_k =$ *k-greedyBFS(T, 2)*

   (b) Compute the expected number of steps to reach a leaf from the root $E[T^{H_g}]$, $E[T^{H_r}]$, and $E[T^{H_k}]$.

   (c) Compute the gain $G(H)$ obtained by each of these algorithms.

[§]Note that the link structure of a web site is different from its file structure.

[¶]Only those that are members of the Association of Universities and Colleges of Canada.

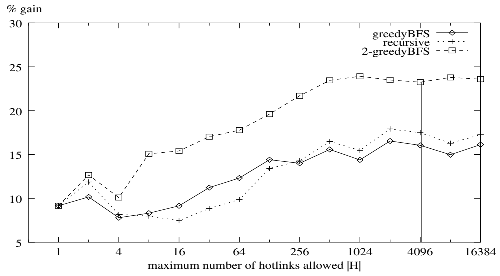

Figure 6: *Average gain obtained by applying* greedyBFS*,* recursive*, and* 2-greedyBFS *to real trees with Zipf's probability distribution over the leaves.*

heuristic	% gain
greedyBFS	16.04
recursive	17.49
2-greedyBFS	23.25

Table 3: *Average gain after applying* greedyBFS*,* recursive*, and* 2-greedyBFS *to real trees with Zipf's probability distribution over the leaves when* $2048 <\mid H \mid \leq 4,096$.

i. $G(H_g) = E[T] - E[T^{H_g}]$
ii. $G(H_r) = E[T] - E[T^{H_r}]$
iii. $G(H_k) = E[T] - E[T^{H_k}]$

We show the results of our simulation in figure 6. Observe that, *greedyBFS* outperforms *recursive* for some values of $\mid H \mid$. This difference in performance is a result of the order in which the recursive calls are made; for example, consider the order in which the recursive calls are made to the subtrees $T_1, T_2, \ldots, T_5$ in figure 4.

The average number of nodes of the web sites sample is 9,669.2. Table 3 shows the average gain when the maximum number of hotlinks allowed per tree is in the range $2048 <\mid H \mid \leq 4096$. It is important to note that it would be unfair to compare the performance of *k-greedyBFS* with the other two algorithms because in *k-greedyBFS* the maximum number of outgoing hotlinks per node is bigger.

5.3 Actual tree with known probability distribution

In this section we describe the way we tested our algorithms on a real web site and then we present our results. We simulated our algorithms on the *scs.carleton.ca* domain. Our simulation was conducted in the following way:

1. Explore the *scs.carleton.ca* domain and obtain its html, htm, shtml, and shtm files.
2. Look at the file (link) references of the home page (*index.html*). Recursively look at the referenced files. Continue in a BFS order until all the files have been visited. This BFS traversal gives rise to a BFS tree T of the web domain.
3. In order to assign access probabilities to the leaves, we took the access log files of the domain. The log files in conjunction are over 57.2 Mbytes, and contain the access logs of 7 days, to a total of 602,879 user accesses. However, we will look only at the accesses with http code 200 or 304 ("OK", and "Not Modified", see [9] for a complete description of the http 1.1 protocol) because they are the only ones that indicate an actual access to the page.
4. For each leaf of the tree T we count the number of hits in the log files. Then we compute the access probability of each leaf, and assign weights (probabilities) to the intermediate nodes in a bottom-up fashion, in such a way that the probability of a node is equal to the sum of the probabilities of the leaves descendant to it.
5. Compute the expected number of steps to reach a leaf from the root $E[T]$.
6. For $i = 1$ to 16384 increment i by $i \times 2$
   (a) Run *greedyBFS*, *k-greedyBFS*, and *recursive* algorithms on the tree with the restriction that $\mid H \mid \leq i$.
      i. $H_g = greedyBFS(T)$
      ii. $H_r = recursive(T)$
      iii. $H_k =$*k-greedyBFS(T, 2)*
   (b) Compute the expected number of steps to reach a leaf from the root $E[T^{H_g}]$, $E[T^{H_r}]$, and $E[T^{H_k}]$.
   (c) Compute the gain $G(H)$ obtained by each of these algorithms.

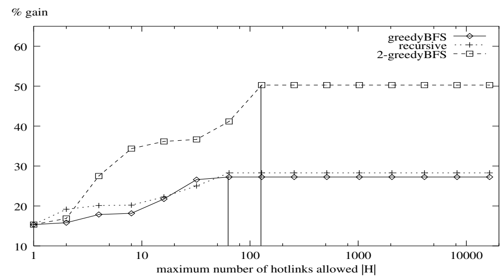

Figure 7: *Gain obtained by applying* greedyBFS*,* recursive*, and* 2-greedyBFS *to the* scs.carleton.ca *domain. Observe that the maximum gain of* greedyBFS *and* recursive *is achieved at* $32 <| H |\leq 64$*, and at* $64 <| H |\leq 128$ *for* 2-greedyBFS.

heuristic	% gain
greedyBFS	27.25
recursive	28.29
2-greedyBFS	50.27

Table 4: *Maximum gain obtained by applying our algorithms to the* scs.carleton.ca *domain.*

$$\begin{aligned} \text{i. } & G(H_g) = E[T] - E[T^{H_g}] \\ \text{ii. } & G(H_r) = E[T] - E[T^{H_r}] \\ \text{iii. } & G(H_k) = E[T] - E[T^{H_k}] \end{aligned}$$

The *scs.carleton.ca* domain contains 798 pages. Figure 7 illustrates the gain obtained in the experiments. Observe that the maximum gain of *greedyBFS* and *recursive* is achieved at $32 <| H |\leq 64 \ll 798$. Table 4 shows the maximum gain of the algorithms. Again, it is important to note that it would be unfair to compare the performance of *k-greedyBFS* with the other two algorithms because in *k-greedyBFS* the maximum number of outgoing hotlinks per node is bigger.

6 Conclusions and future research

We have introduced three hotlink assignment heuristic algorithms for improving web access. We described how to create a tree from a web site. Such a tree maintains the original designer's visualization of the site. We evaluated our algorithms on random and real trees with Zipf's distribution, and on an actual tree with a known distribution. The simulation revealed that we can save at least 17%, and as much as 28% of the original access cost of a web site.

This paper has been an introduction to more complex hotlink assignment algorithms that we believe will outperform the ones that we have presented here. We are envisioning algorithms that promise to approach the problem in a more elegant way. Some of them are divide-and-conquer oriented, while others use dynamic programming techniques.

References

[1] J. Abello, P.M. Pardalos, and M.G.C. Resende. On Maximum Clique Problems in Very Large Graphs. *In External Memory Algorithms (J. Abello and J. Vitter, eds.), AMS-DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 50, 1999.

[2] David Abrams. How People Use WWW Bookmarks. In *Conference on Human Factors in Computing Systems*, Atlanta, GA, May 1997. http://www.acm.org/sigchi/chi97/proceedings/short-talk/da.htm.

[3] Prosenjit Bose, Jurek Czyzowicz, Leszek Gasieniec, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, and Miguel Vargas Martin. Strategies for Hotlink Assignments. In *Proceedings of the Eleventh Annual International Symposium on Algorithms and Computation (ISAAC 2000)*, pages 23–34, Taipei, Taiwan, December 2000.

[4] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph Structure in the Web. In *Proceedings of the Nineth International WWW Conference*, Amsterdam, Holland, May 2000. http://www9.org/w9cdrom/160/160.html.

[5] Lara D. Catledge and James E. Pitkow. Characterizing Browsing Strategies in the World-Wide Web. In *Proceedings of the Third International World-Wide Web*

Conference, Darmstadt, Germany, April 1995. http://www.igd.fhg.de/archive/1995_ www95/papers/80/userpatterns/UserPattern-s. Paper4.formatted.html.

[6] M. Carl Drott. Using Web Server Logs to Improve Site Design. In *ACM 16th Annual International Conference of Computer Documentation (SIGDOC-98)*, Getting feedback on your Web site, pages 43–50, New York, NY, September 23–26 1998. ACM Press.

[7] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law Relationships of the Internet Topology. *Computer Communication Review*, 29(4):251–262, 1999.

[8] Steven Glassman. A Caching Relay for the World Wide Web. http://www.research.digital.com/SRC/ people/Glassman_Steve/ CachingTheWeb.ht-ml, 1994.

[9] Network Working Group. Request for Comments: 2068. http://www.cis.ohio-state.edu/htbin/rfc/rfc2068.html, 1997.

[10] Brian Hayes. Graph Theory in Practice: Part I. *American Scientist*, 88(1), January–February 2000.

[11] Brian Hayes. Graph Theory in Practice: Part II. *American Scientist*, 88(2), March–April 2000.

[12] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[13] Bo Li, M. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the Optimal Placement of Web Proxies in the Internet. In *Proceedings of the IEEE InfoCom Conference*, March 1999.

[14] Ari Luotonen and Kevin Altis. World-Wide Web Proxies. In *Proceedings of the First International Conference on the World-Wide Web*, Geneva, Switzerland, April 1994. http://www1.cern.ch/PapersWWW94/ luotonen.ps.

[15] Mike Perkowitz and Oren Etzioni. Adaptive Sites: Automatically Learning from User Access Patterns. In *Proceedings of the Sixth International WWW Conference*, Santa Clara, CA, April 1997. ftp://ftp.cs.washington.edu/tr/1997/03/UW-CSE-97-03-01.PS.Z.

[16] Mike Perkowitz and Oren Etzioni. Towards Adaptive Web Sites: Conceptual Framework and Case Study. In *Proceedings of the Eigth International WWW Conference*, Toronto, Canada, May 1999. http://www8.org/w8-papers/2b-customizing/towards/towards.html.

[17] James E. Pitkow. Summary of WWW Characterizations. Technical report, Xerox Palo Alto Research Center, 1997.

[18] James E. Pitkow and Margaret M. Recker. A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, Atlanta, GA, 1994. http://www.ncsa.uiuc.edu/SDG/IT94/ Proceedings/DDay/pitkow/caching.html.

[19] Albert Réka, Jeong Hawoong, and Albert-László Barabási. Diameter of the World-Wide Web. *Nature*, 401:130–131, 1999.

[20] Linda Tauscher and Saul Greenberg. Revisitation Patterns in World Wide Web navigation. In *Conference on Human Factors in Computing Systems*, Atlanta, GA, May 1997. http://www.acm.org/sigs/sigchi/chi97/ proceedings/paper/sg.htm.

[21] Ellen W. Zegura, Kenneth L. Calvert, and Michael J. Donahoo. A Quantitative Comparison of Graph-Based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, 5(6):770–783, December 1997.